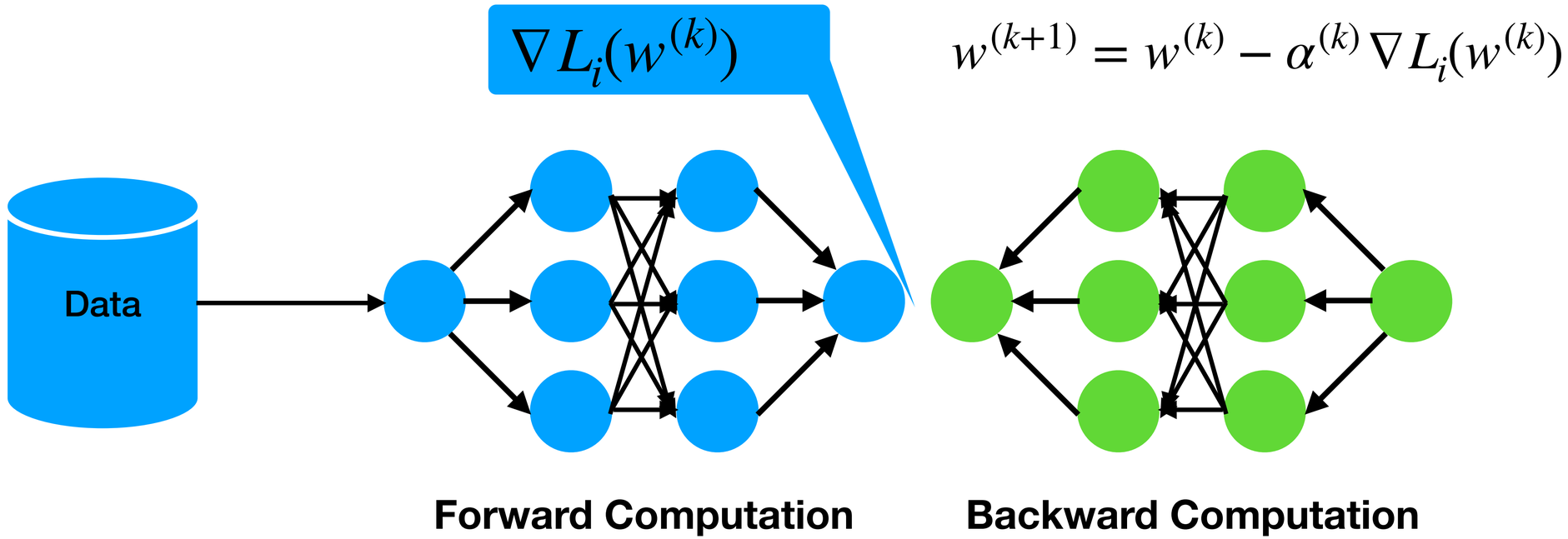
Distributed Neural Network Training

Zhao Zhang
Department of Electrical and Computer Engineering
Rutgers University
Oct 7th, 2025

Frontera Supercomputer

- Primary compute system:
  - 39PF PetaFlops Peak Performance -- 8,000+ nodes of Intel Cascade Lake
- Storage: DataDirect Networks
  - 50+ PB disk, 3PB of Flash, 1.5TB/sec peak I/O rate.
- Front end for data movers, workflow, API
- GPU Subsystem:
  - 90 node with four RTX5000 GPU each

Recap of Neural Network Training



Why Distributed Training

- Data is too big
  - ImageNet has 1.2 M images
  - The PILE dataset has 800 GB text
  - OpenProteinSet has ~1 TB protein chains, clusters, and multi-sequence alignments
  - GPT-3 is trained with 45 TB text

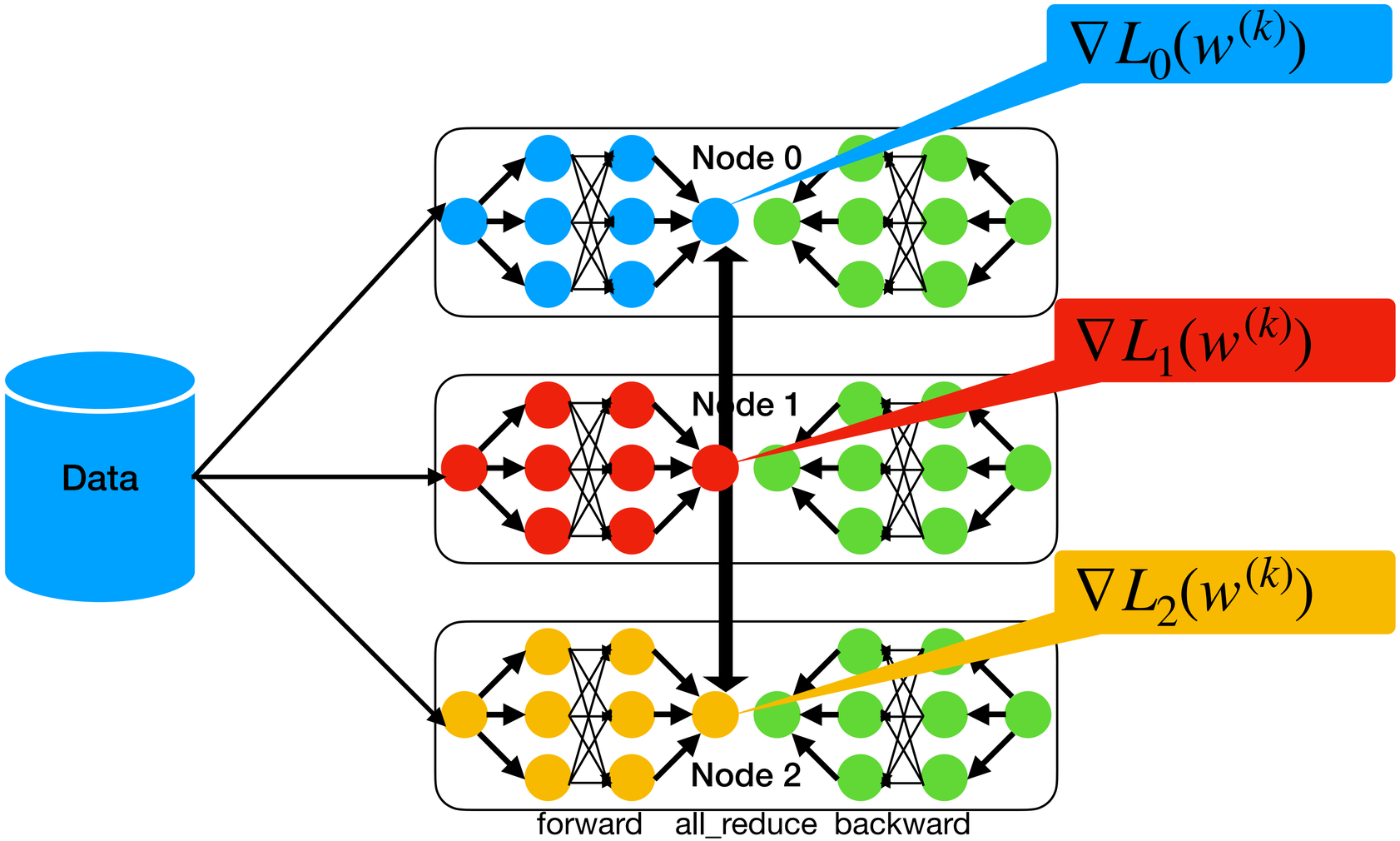
Why Distributed Training

- Training is too long
  - OPT-175B takes 1,024 A100 GPUs for 2 months
  - OpenFold takes 128 A100 GPUs for 11 days
  - GPT-NeoX 20B takes 96 A100 GPUs for 30 days
  - ViT takes 1,960 GPU hours (A100, 40G)
- You don’t just run it once
  - Hyper-parameter tuning
  - Architecture search

Why Distributed Training

- Model is too big
  - AlphaFold2 has 21 M parameters
  - BERT has 345 M parameters
  - ViT Huge has 632 M parameters
  - LLaMA-2 has 7, 13, and 70 B parameters
  - GPT-3 has 175 B parameters
- A100 and H100 has 80 GB HBM versions

Data Parallelism



- **Parameters (Model) are duplicated on all nodes**

Data Parallelism

- Is it as this simple and easy?
- Parameter Server

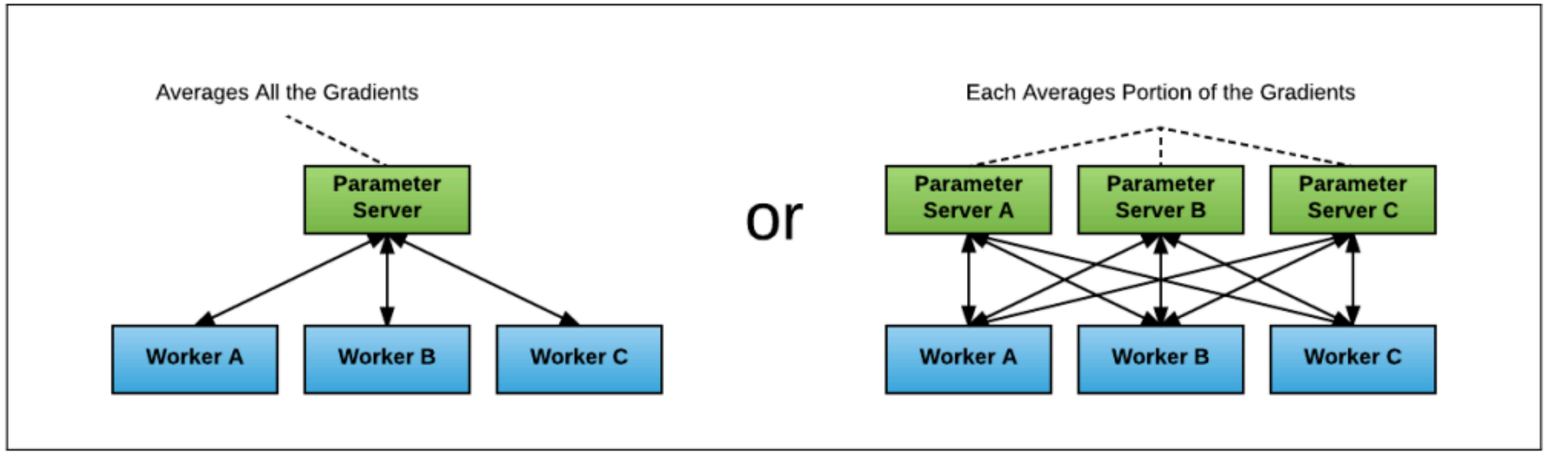
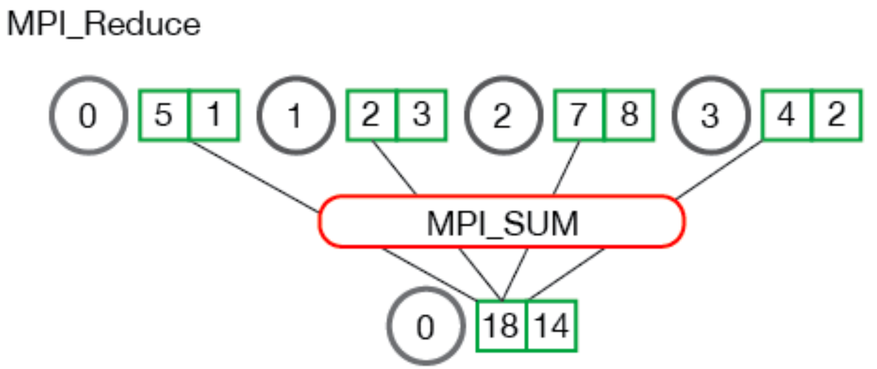


Figure 3: The parameter server model for distributed training jobs can be configured with different ratios of parameter servers to workers, each with different performance profiles.

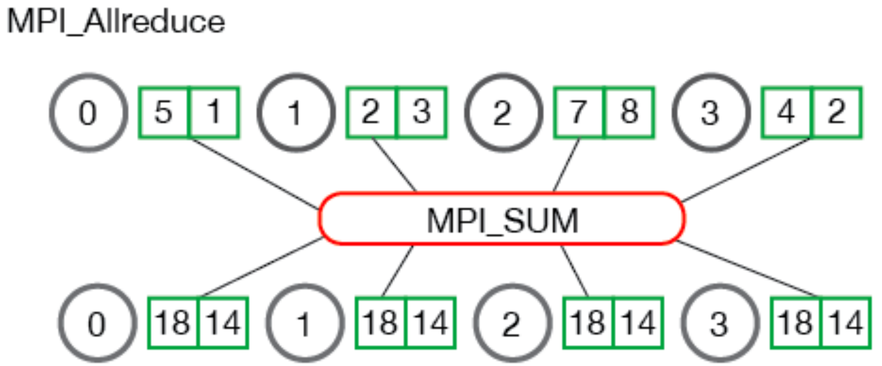
- All to all communication among processes to compute the sum of the gradients

Averaging Gradients

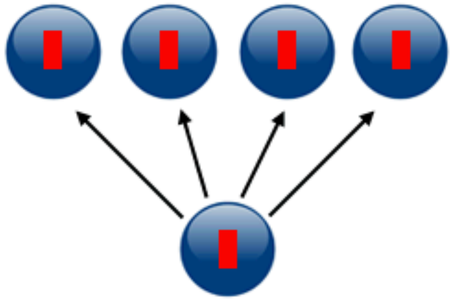
- Reduce



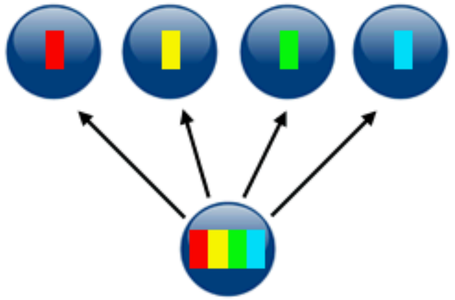
- All-Reduce



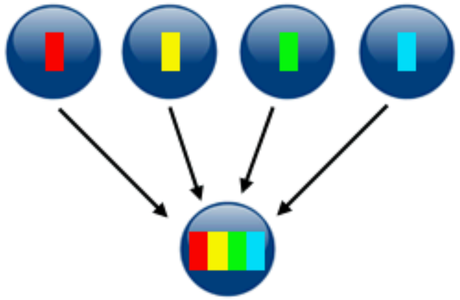
Collective Operations



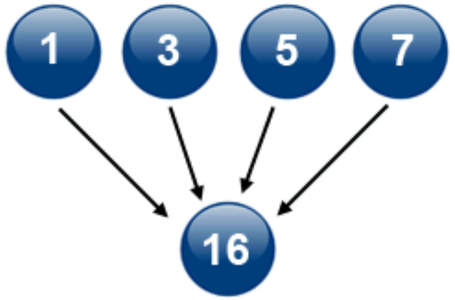
broadcast



scatter



gather



reduction

AllReduce

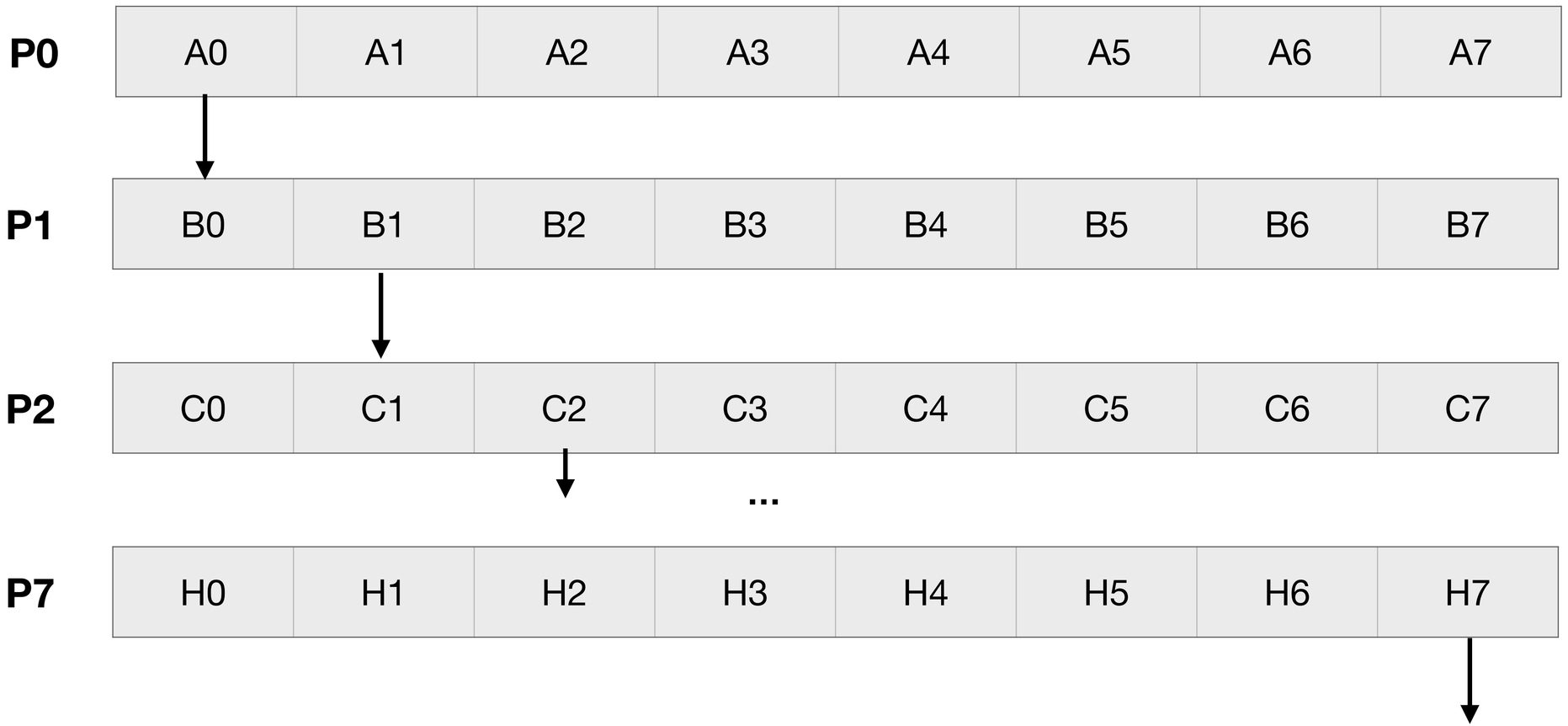
- Single buffer or Chunked buffer
  - Ring
  - 2D-torus
  - Recursive Doubling

AllReduce

- Reduce + Scatter
  - Ring Topology
  - Every process communicates with its neighbors

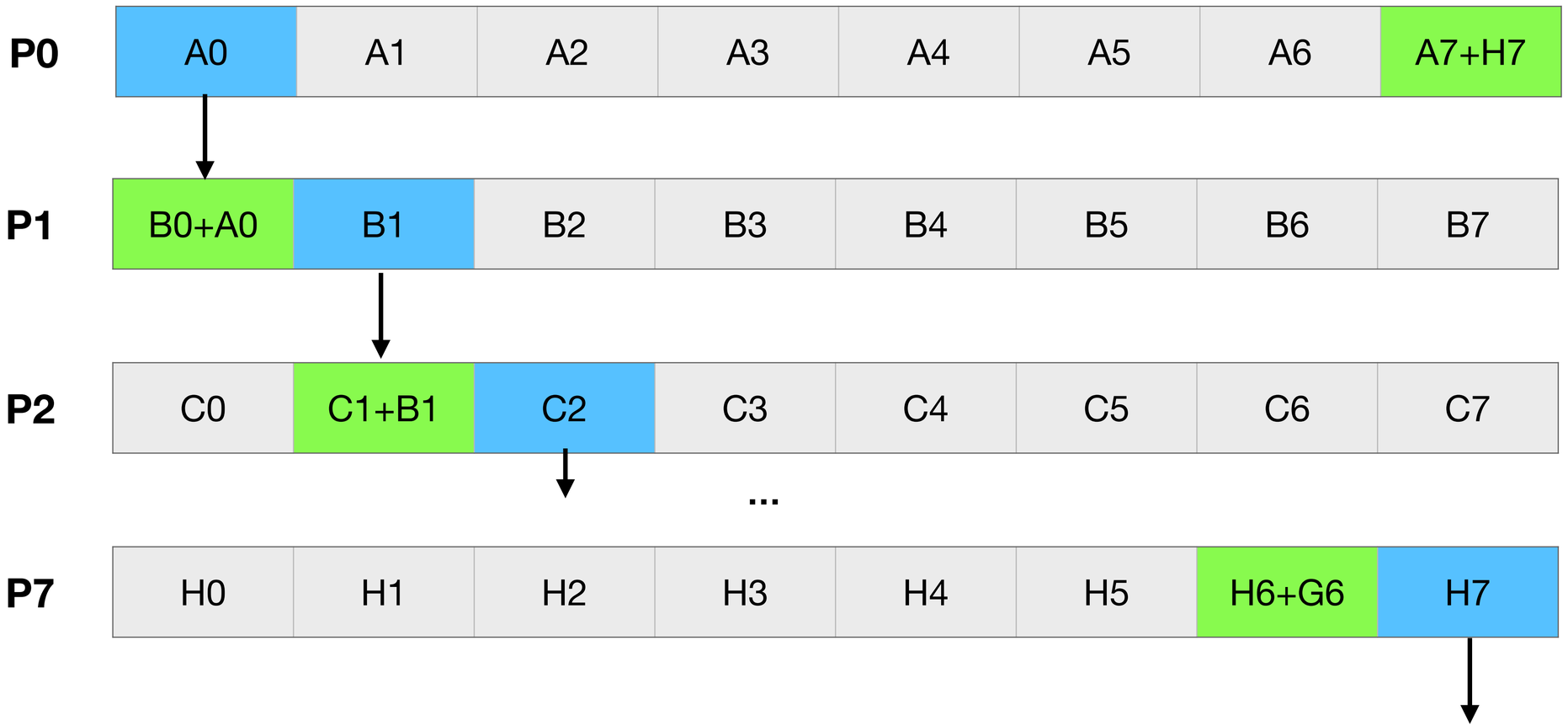
AllReduce — Reduce Phase

- Step 1



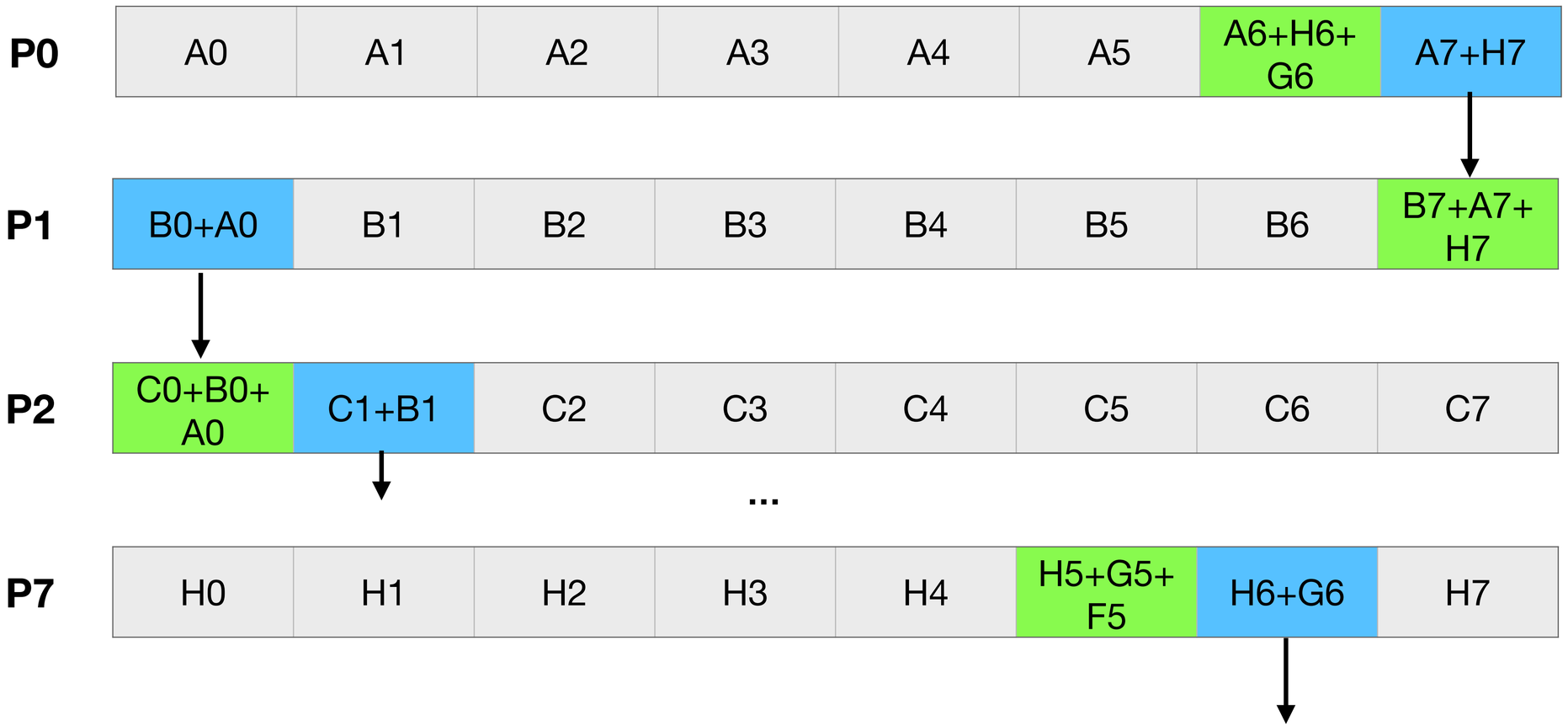
AllReduce — Reduce Phase

- Step 1



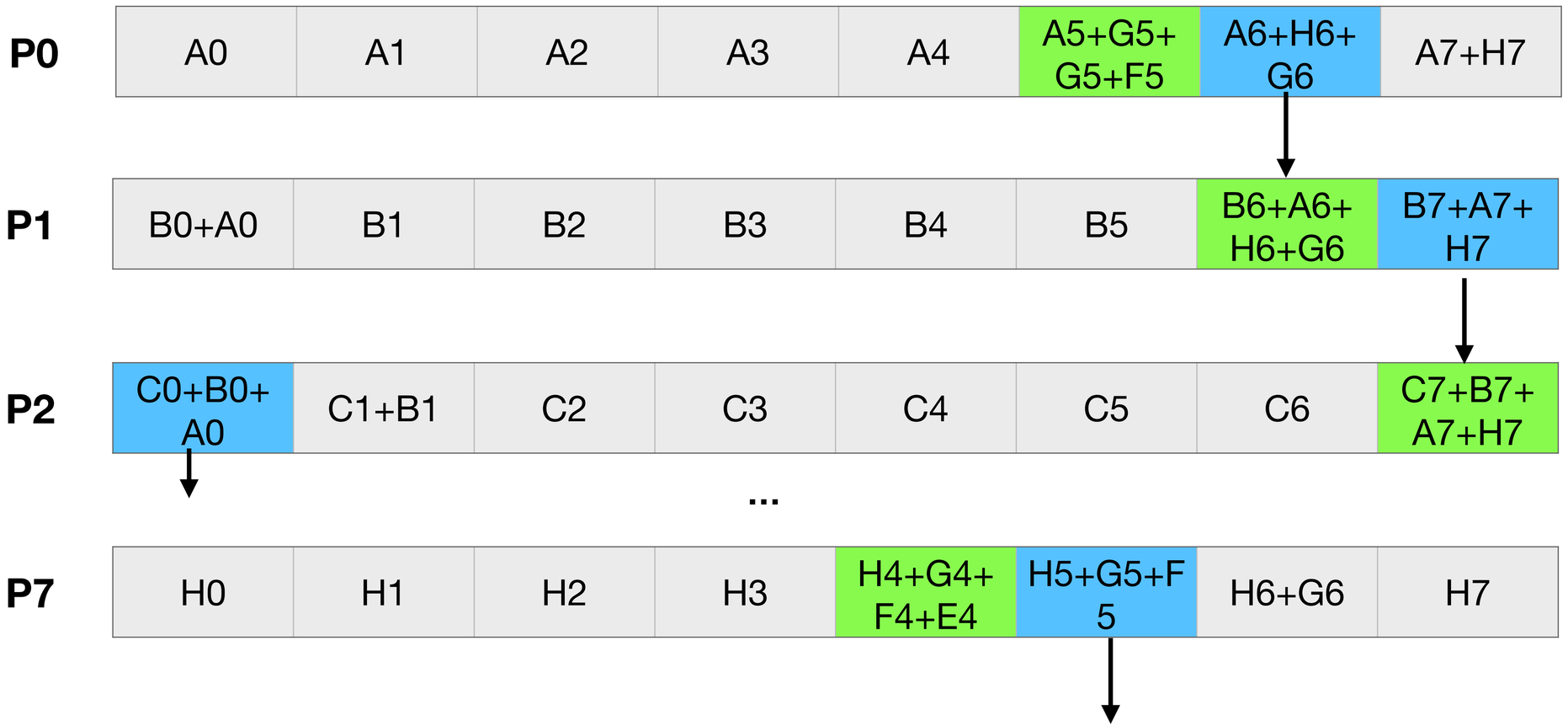
All Reduce — Reduce Phase

- Step 2



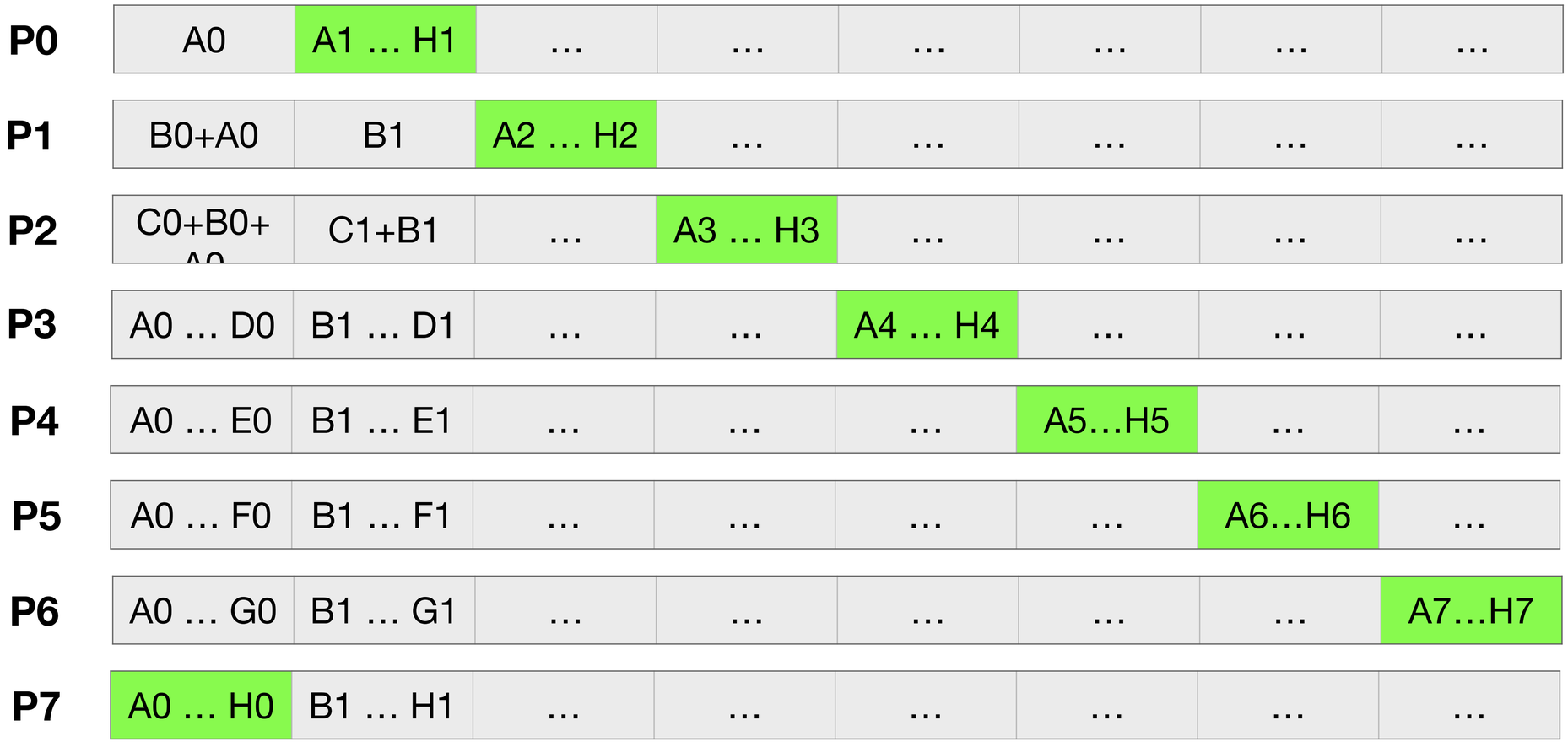
AllReduce — Reduce Phase

- Step 3



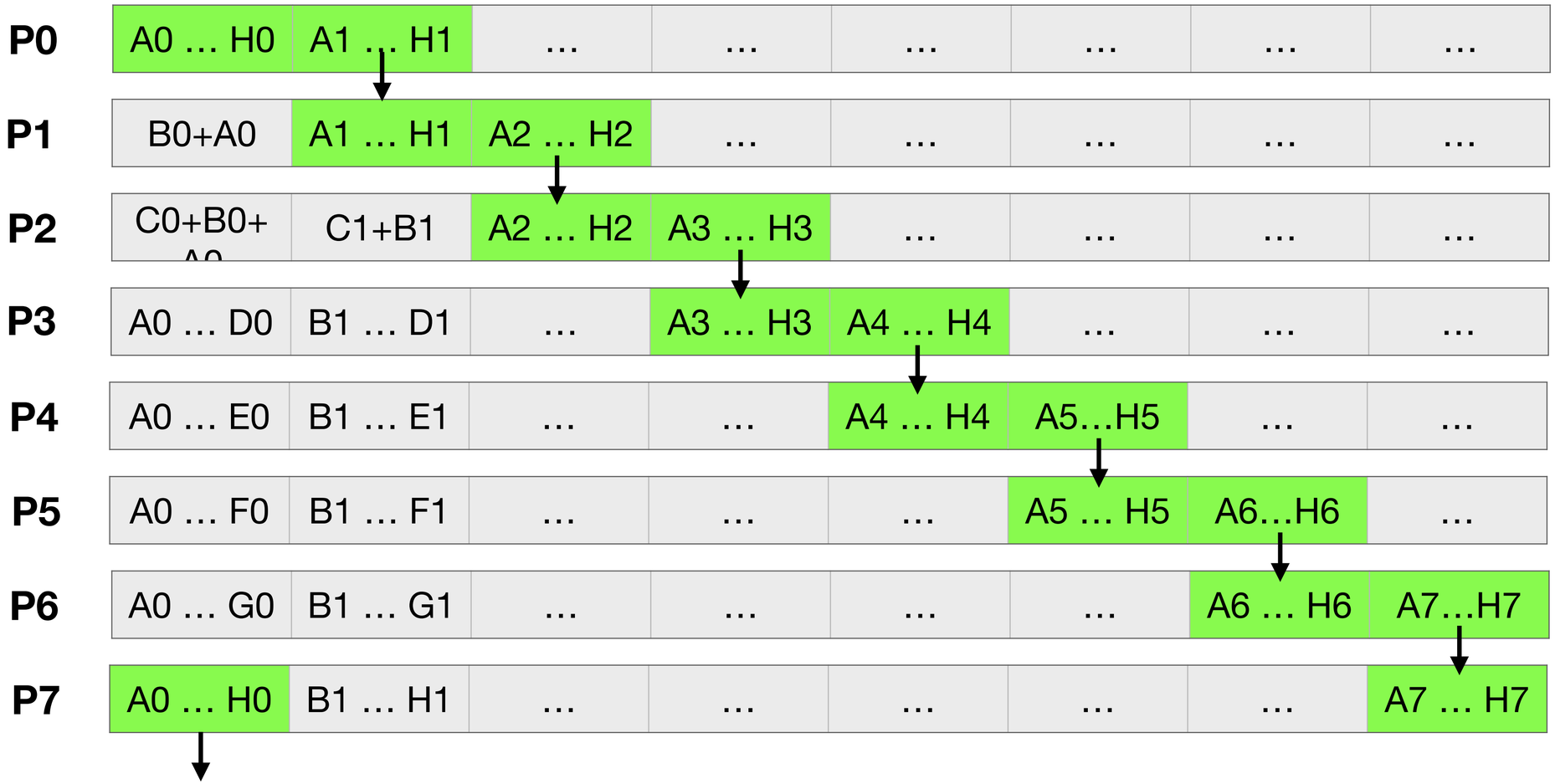
AllReduce — Reduce Phase

- Step 7



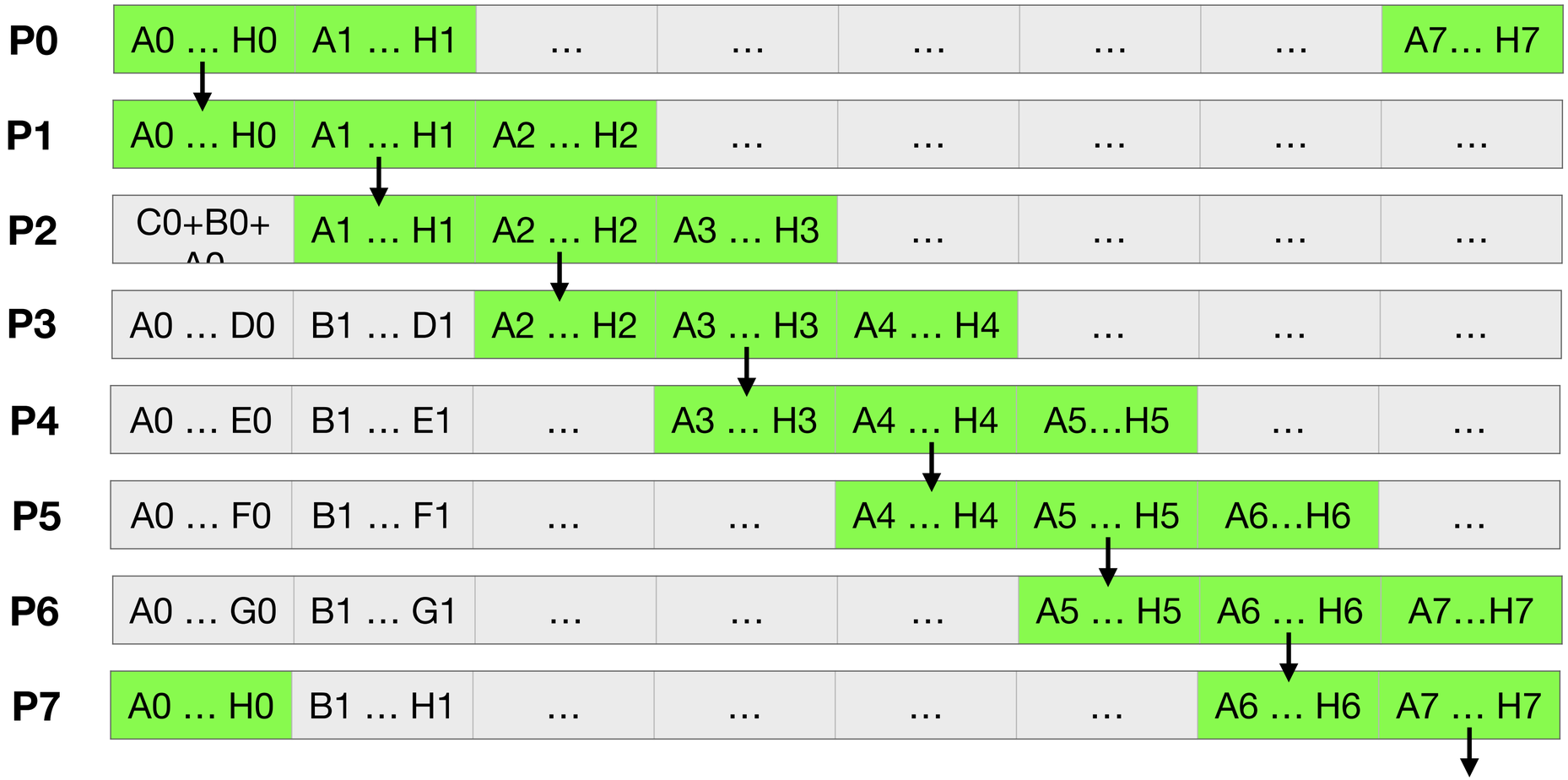
AllReduce — Scatter Phase

- Step 1



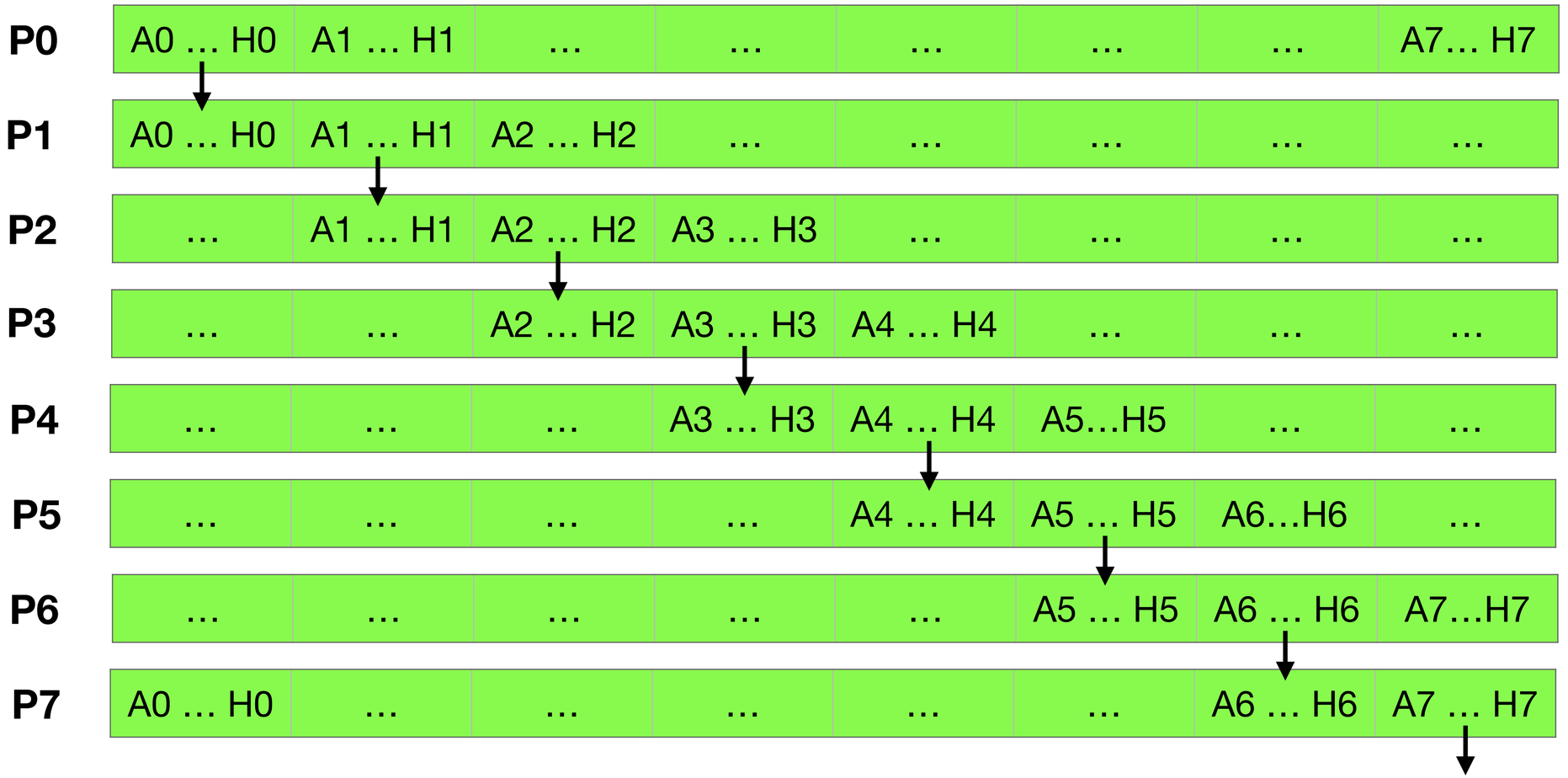
AllReduce — Scatter Phase

- Step 2



AllReduce — Scatter Phase

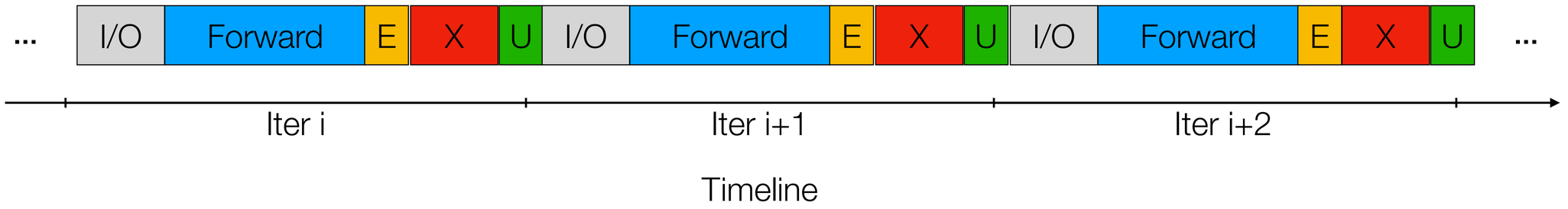
- Step 7



AllReduce — Scatter Phase

- Complexity
  - (N-1) Reduce Steps
  - (N-1) Scatter Steps
  - Each step takes $\alpha + B \times \beta$
  - $T = 2(N-1)(\alpha + B\beta) = 2(N-1)(\alpha + \frac{1}{N}D\beta) = 2(N-1)\alpha + 2\frac{N-1}{N}D\beta$

Execution Model

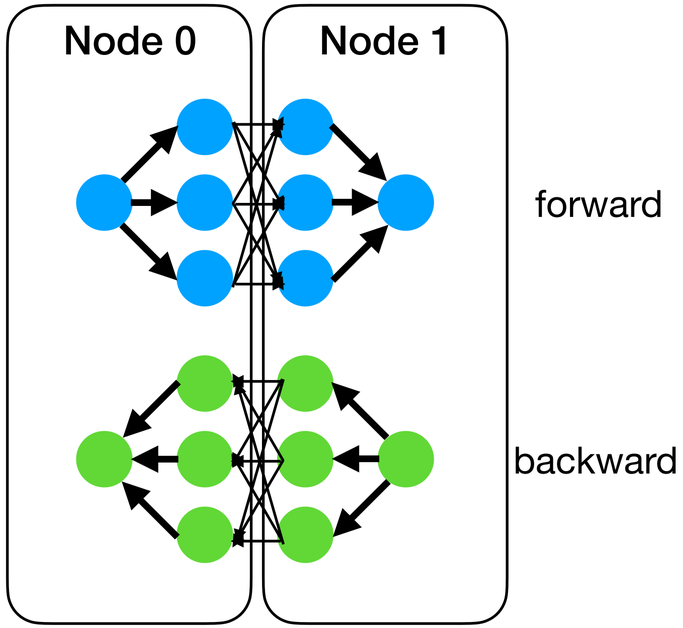


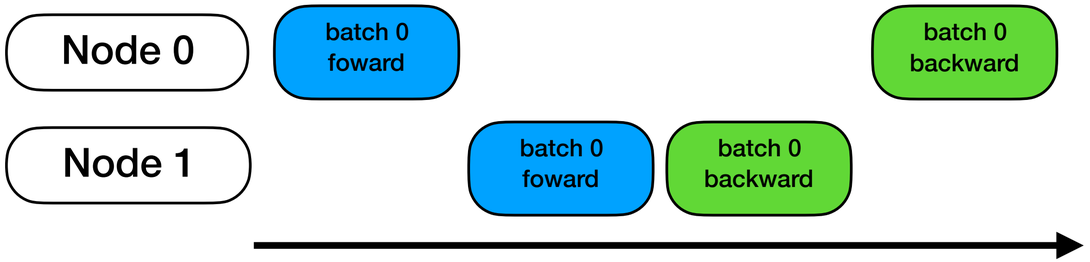
- A simple understanding:

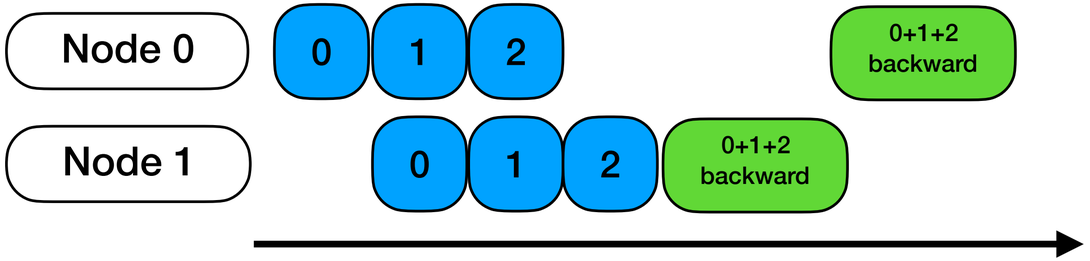
$$T = T_{forward} + T_{allreduce}$$

Faster Interconnect
Faster Algorithm
Less Data

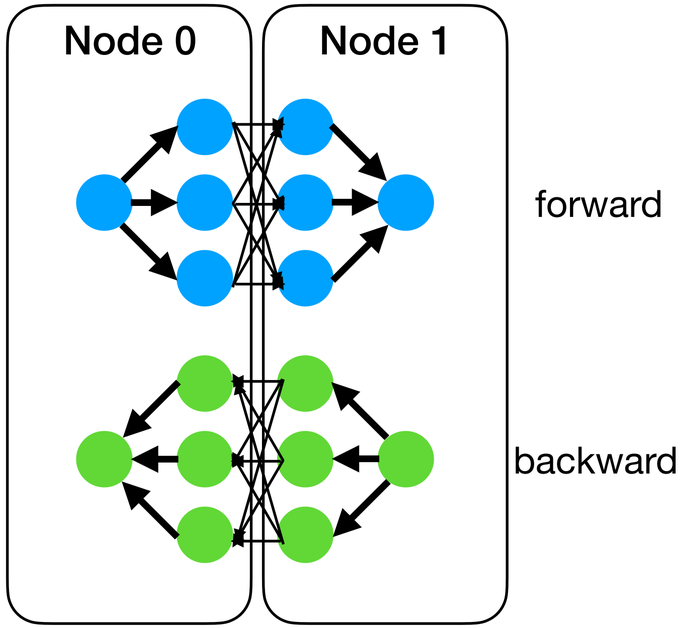
Model Parallelism

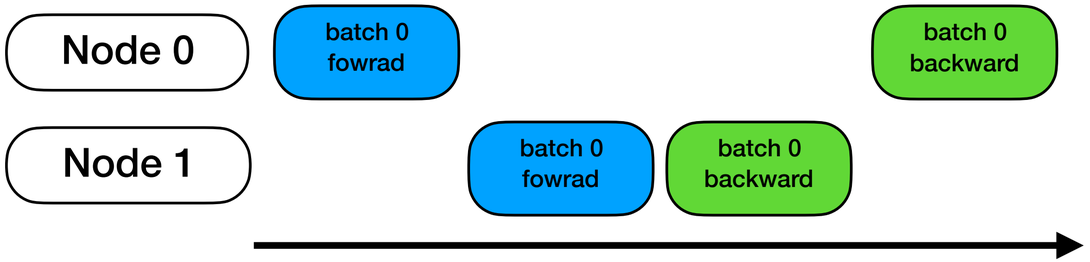


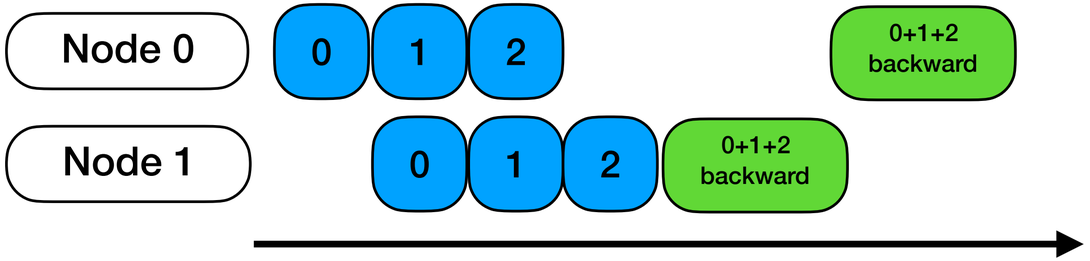




Pipeline Parallelism

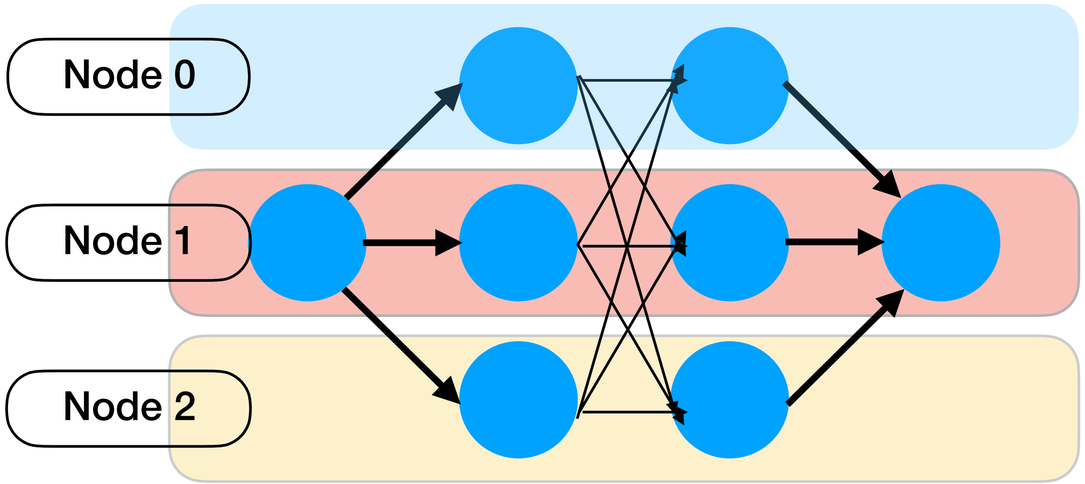




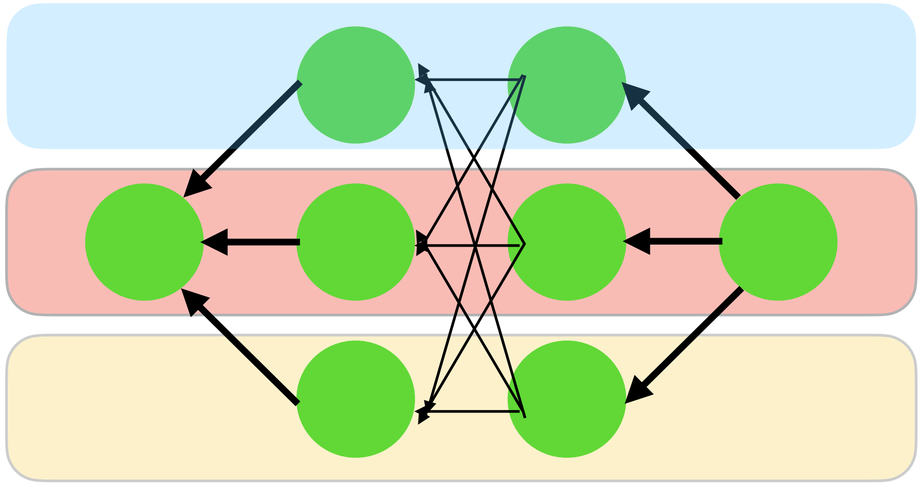


Tensor Parallelism

- What are being communicated?

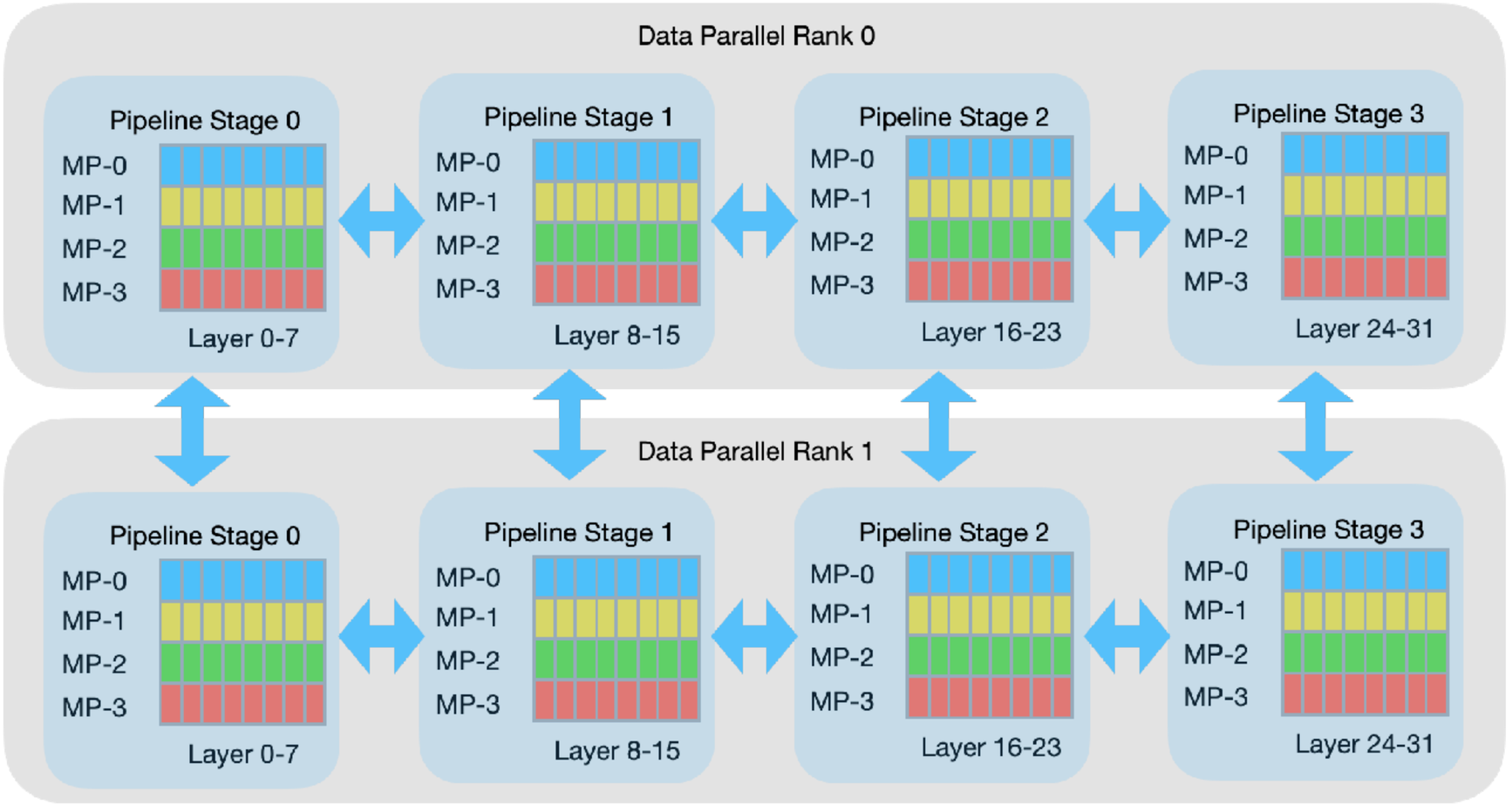


Forward Computation

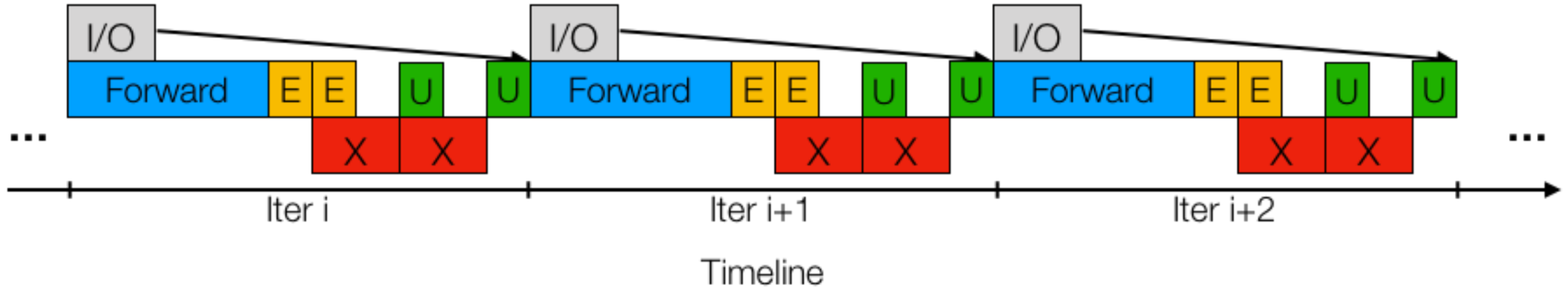


Backward Computation

3D Parallelism



Execution Model



- A simple understanding:

$$T = T_{forward} + \sum T_{allreduce}$$

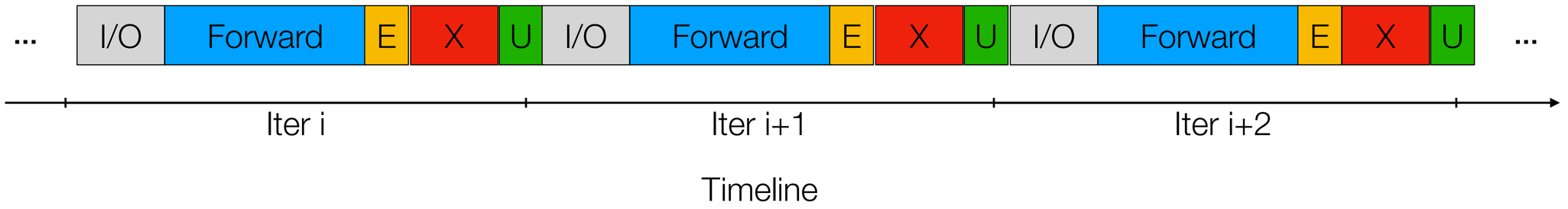
Faster Computation

Faster Interconnect
Faster Algorithm
Less Data

Scaling Efficiency

- Strong Scaling Efficiency
  - Fixed Problem
    - Fixed Model and Fixed Global Batch Size
  - $P_{scale}/(P_{baseline} \times scale)$
- Weak Scaling Efficiency
  - Problem Size in Proportional to Scale
    - Fixed Model and Linearly Scaling Global Batch Size
  - $P_{scale}/(P_{baseline} \times scale)$

Execution Model



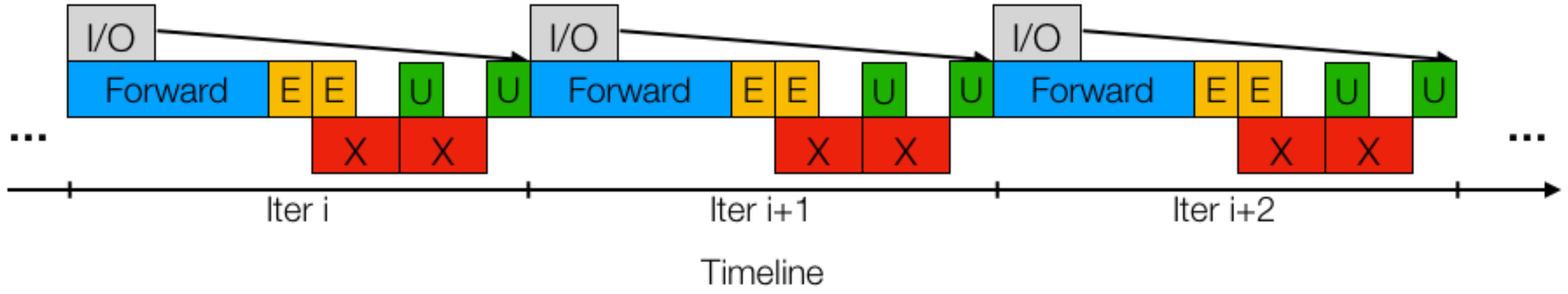
- A simple understanding:

$$T = T_{forward} + T_{allreduce}$$

Fast Computation

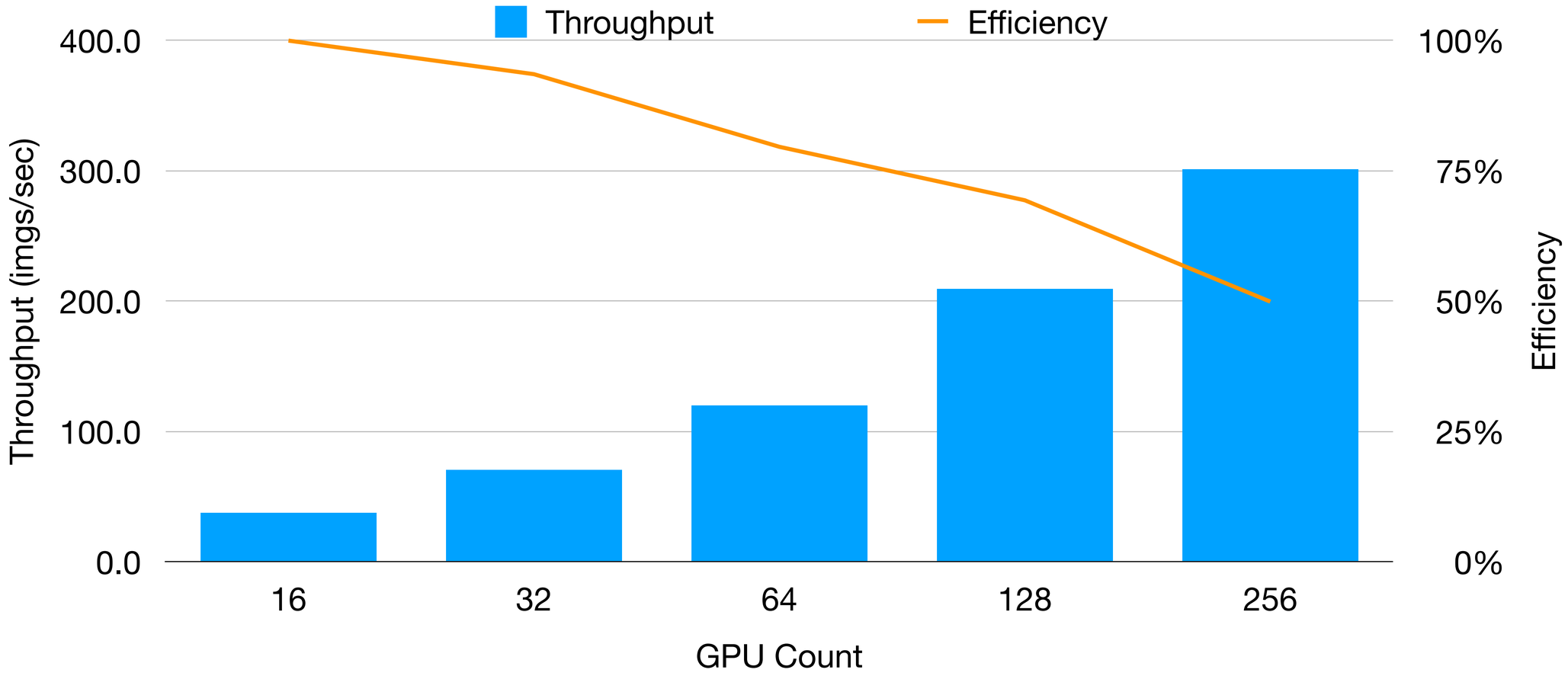
Faster Interconnect
Faster Algorithm
Less Data

Performance Analysis

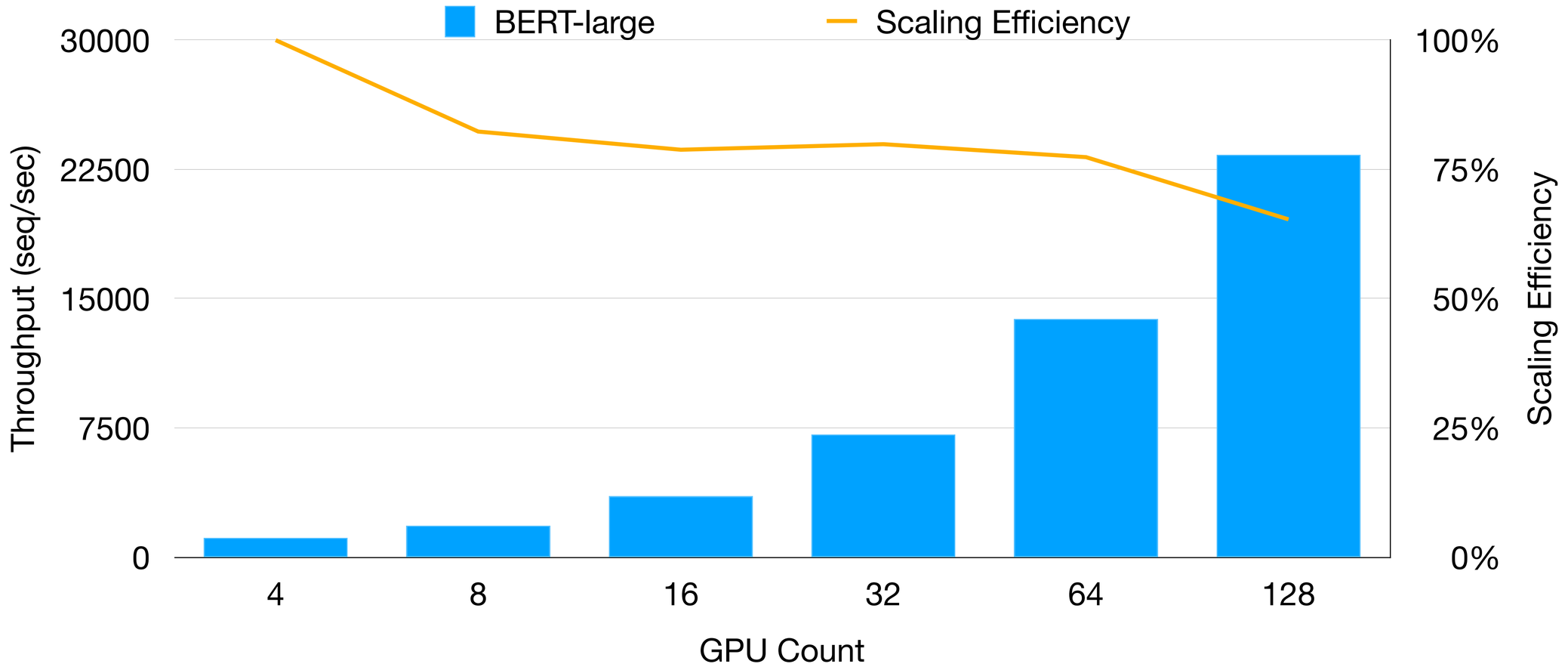


- Fixed Model and Batch Size per GPU, Larger Scale
- Fixed Model and Larger Batch Size per GPU, Fixed Scale
- Larger Model and Fixed Batch Size per GPU, Fixed Scale

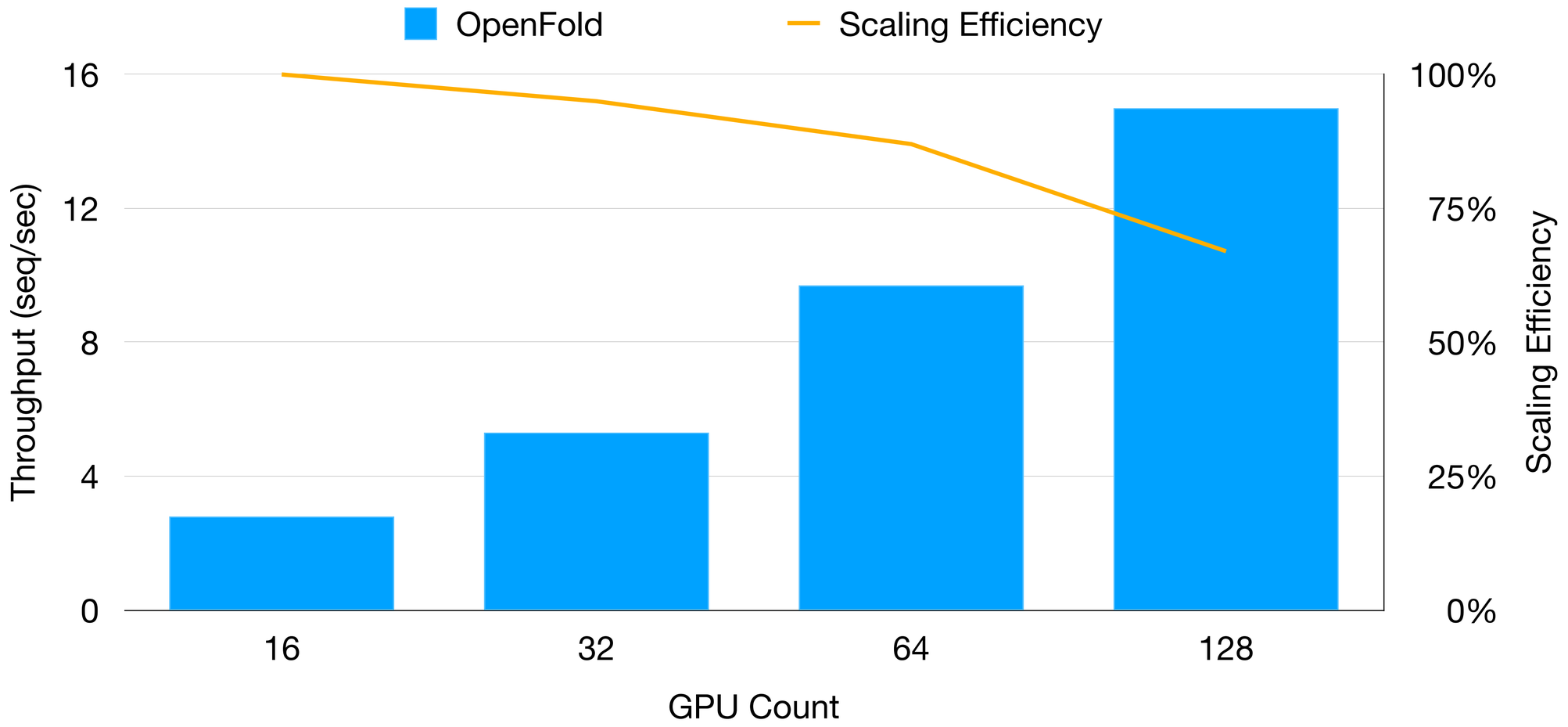
ResNet-50 Example



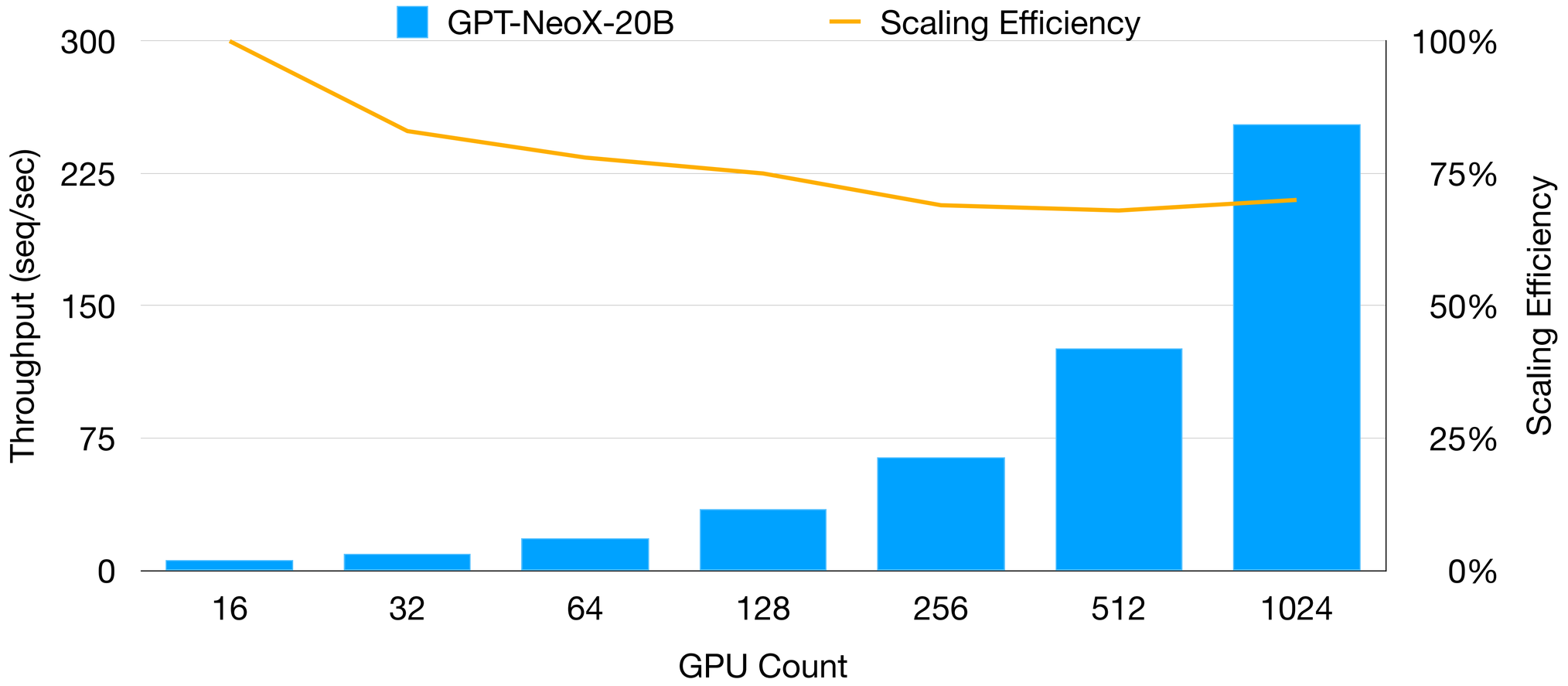
BERT Example



OpenFold Example

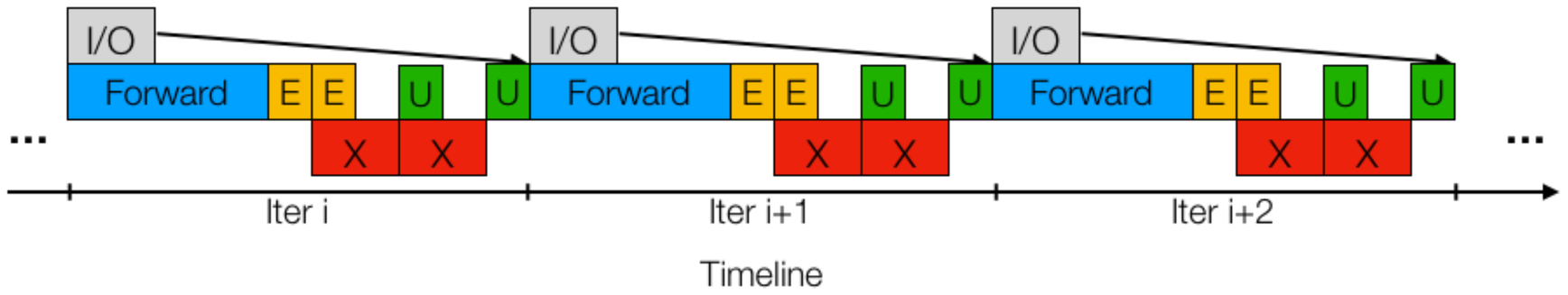


GPT-NeoX Example



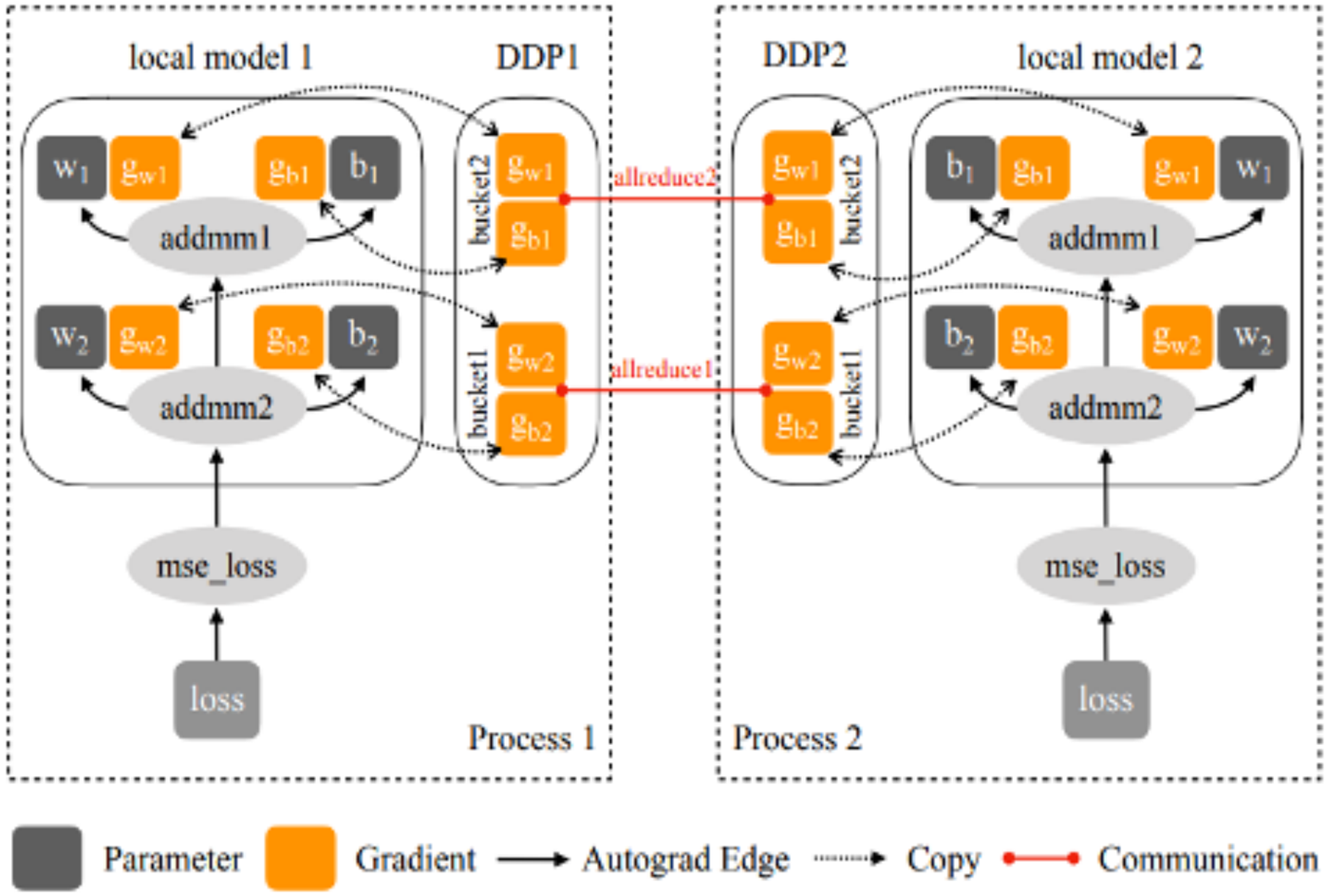
Case Study: PyTorch

- The AllReduce operation dose not need to wait for the finishing of backward propagation.
  - AllReduce for each gradient is independent.
  - Backward propagation is a serial computation. Getting the gradients of last layer at first and that of first layer at last.
- Backward propagation and AllReduce operation can be run at same time.



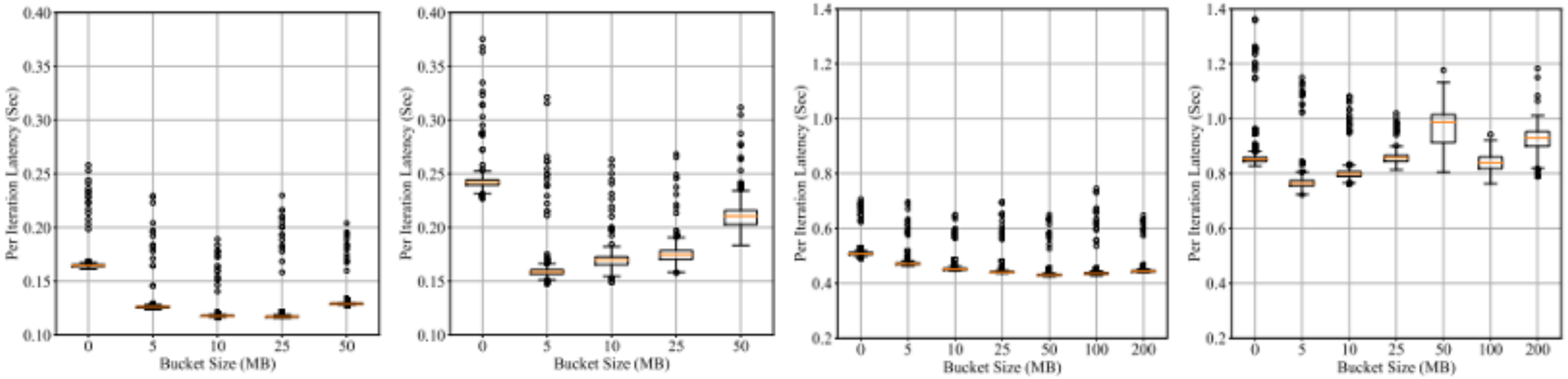
Case Study: PyTorch

- Gradient Bucket
  - It is not efficient to do AllReduce for each parameter's gradients separately.
    - Too many times of communication
    - Can not leverage bandwidth and computing power
  - Doing AllReduce for a bucket of gradients at every time.



Case Study: PyTorch

- Latency vs. Bucket Size



(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

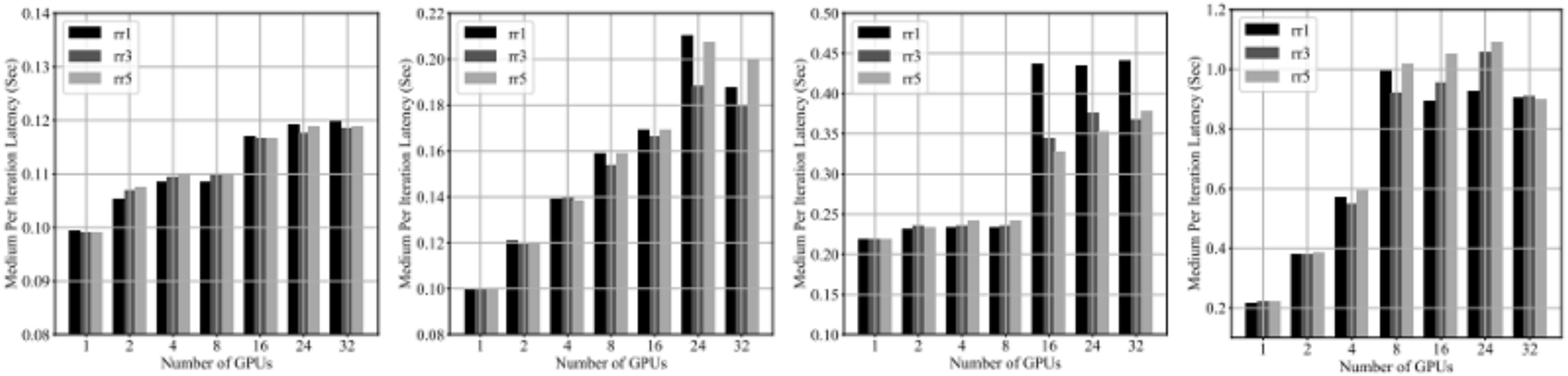
Figure 7: Per Iteration Latency vs Bucket Size on 16 GPUs

Case Study: PyTorch

- Process Group
  - Point-to-point communication for the process with each others is not efficient.
    - Divide processes to smaller groups
    - Round-robin manner (P2P) across groups

Case Study: PyTorch

- Process Group

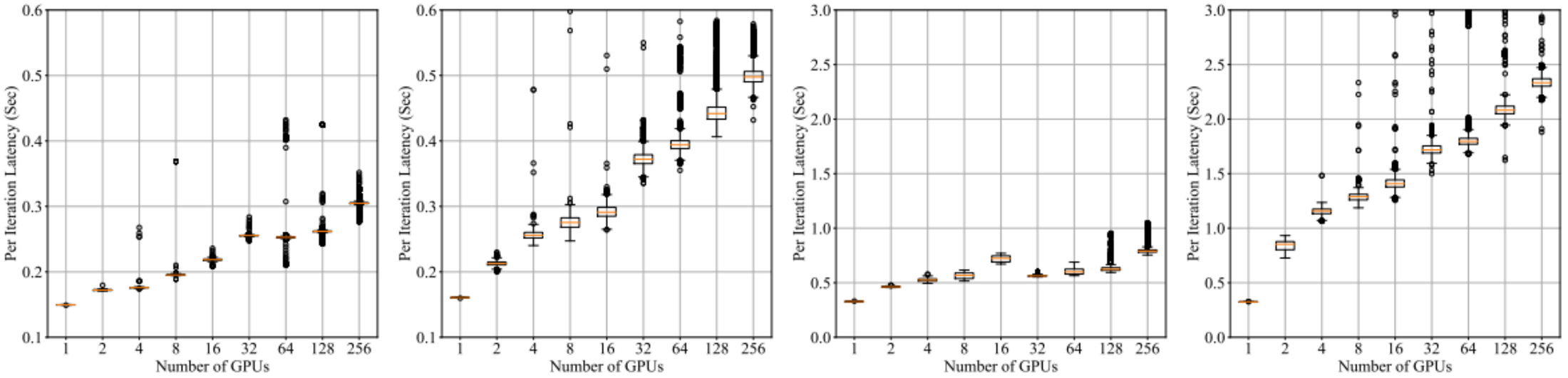


(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

Figure 12: Round-Robin Process Group

Case Study: PyTorch

- Scalability



(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

Figure 9: Scalability

Case Study: PyTorch

- Main purpose: Optimizing latency for strategy of Data Parallelism & Synchronous SGD.
  - Method 1: Overlap computation (backward propagation & AllReduce) and gradient bucket
  - Method 2: Process group

PyTorch Tutorial

- You can compose a PyTorch program in four steps:
  - Dataset Preparation
  - Model Definition
  - Optimizer Specification
  - Training Instrumentation

PyTorch Dataset

- Dataset —> Dataloader

- PyTorch has built-in datasets, e.g., CIFAR10

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose

train_data = datasets.CIFAR10(root="/tmp", train=True, download=True, transform=ToTensor())

test_data = datasets.CIFAR10(root="/tmp", train=False, download=True, transform=ToTensor())

batch_size = 128

# Create data loaders.
train_dataloader = DataLoader(train_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
```

PyTorch Model — nn.Sequential()

```
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = Net().to(device)
```

PyTorch Model — nn.Functional()

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = Net().to(device)
```

PyTorch Optimizer

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

PyTorch Training

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

PyTorch Training

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
                        y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

PyTorch Training

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print(“Done!”)
```

Distributed PyTorch Tutorial

```
import torch.distributed as dist

def main():
    ...
    torch.distributed.init_process_group(backend='nccl', init_method='env://')
    ...
    torch.cuda.set_device(args.local_rank)
    ...
    model = torch.nn.parallel.DistributedDataParallel(model,
            device_ids=[args.local_rank])
    ...
    train_sampler = torch.utils.data.distributed.DistributedSampler(
            train_dataset, num_replicas=args.backend.size(), rank=args.backend.rank())
    train_loader = torch.utils.data.DataLoader(
            train_dataset, batch_size=args.batch_size * args.batches_per_allreduce,
            sampler=train_sampler, **kwargs)

    val_sampler = torch.utils.data.distributed.DistributedSampler(
            val_dataset, num_replicas=args.backend.size(), rank=args.backend.rank())
    val_loader = torch.utils.data.DataLoader(
            val_dataset, batch_size=args.val_batch_size,
            sampler=val_sampler, **kwargs)
    ...

    for epoch in epochs:
        train()
        test()
```

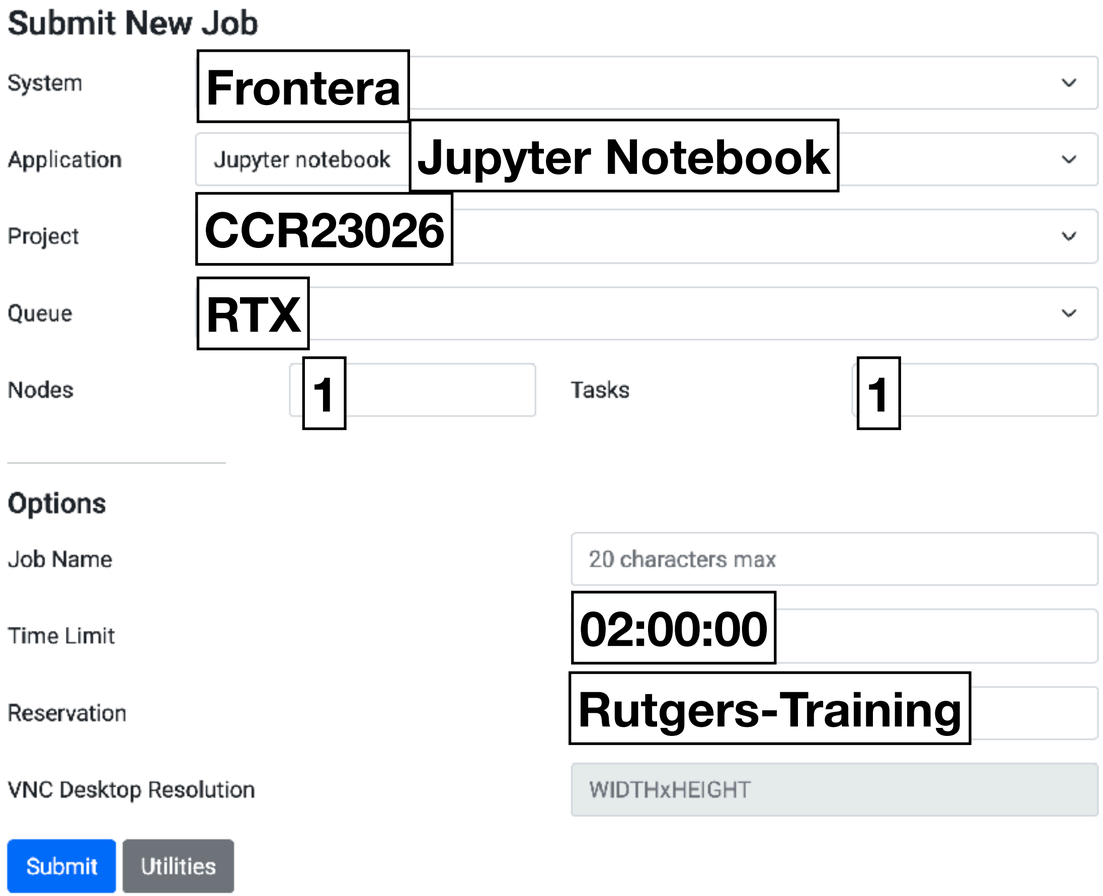
Distributed PyTorch Tutorial

- One node with four GPUs

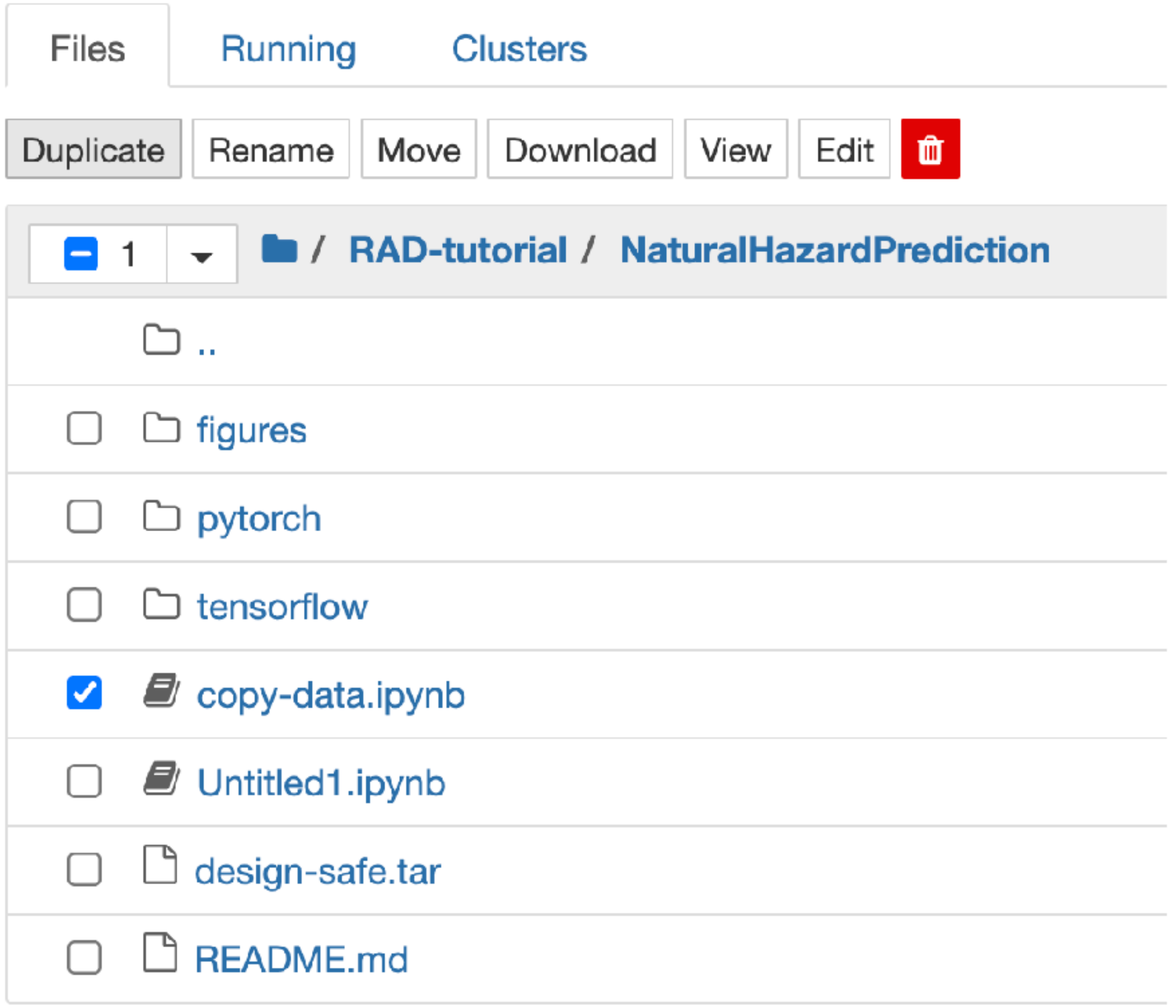
```
python -m torch.distributed.launch --nproc_per_node=4
    kfac_pytorch/examples/torch_cifar10_resnet.py \
        --data-dir /tmp/cifar-10-batches-py \
        --base-lr 0.1 \
        --batch-size 128 \
        --epochs 100 \
        --kfac-update-freq 0 \
        --model resnet32 \
        --lr-decay 35 75 90
```

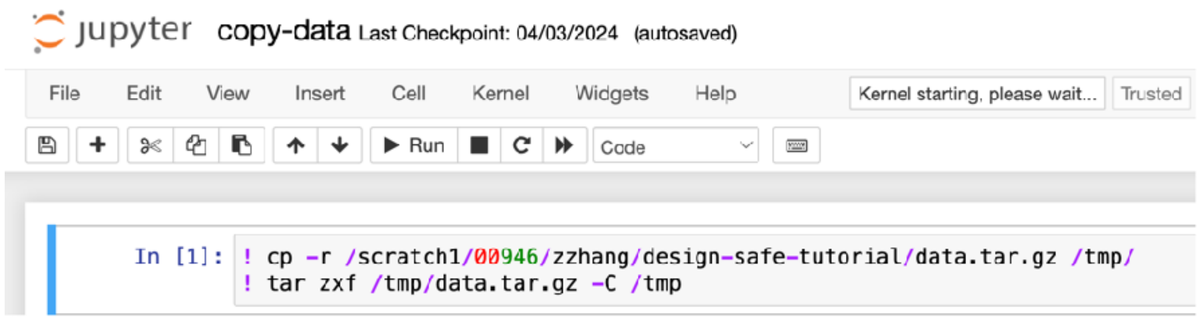
Jupyter Notebook

- https://tap.tacc.utexas.edu/

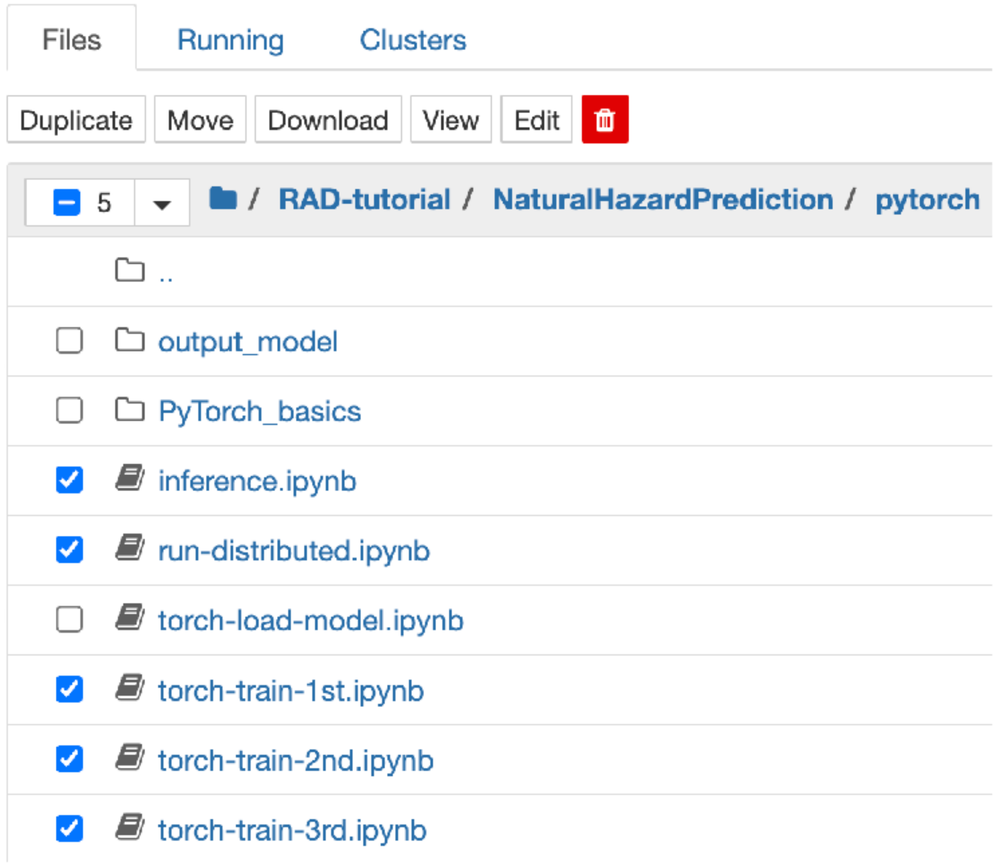


Copy Data


Files
Running
Clusters
Duplicate
Rename
Move
Download
View
Edit
1
/ RAD-tutorial / NaturalHazardPrediction
..
figures
pytorch
tensorflow
copy-data.ipynb
Untitled1.ipynb
design-safe.tar
README.md


jupyter copy-data Last Checkpoint: 04/03/2024 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Kernel starting, please wait...
Trusted
Run
Code
In [1]:
! cp -r /scratch1/00946/zzhang/design-safe-tutorial/data.tar.gz /tmp/
! tar zxf /tmp/data.tar.gz -C /tmp

Exercise



- Run torch-train-1st.ipynb
- Run torch-train-2nd.ipynb
- Run torch-train-3rd.ipynb
- Run run-distributed.ipynb
- Run inference.ipynb
  - Run the last cell repeatedly